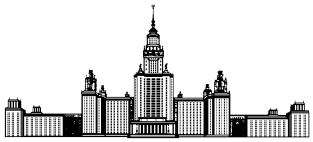
Московский Государственный Университет им. М.В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Математических Методов Прогнозирования

Курсовая работа

«Регуляризация тематических моделей в библиотеке BigARTM»

Выполнил:
студент 3 курса 317 группы
Апишев Мурат Азаматович

Научный руководитель:
д.ф-м.н., доцент
Воронцов Константин Вячеславович.

Москва, 2014

Содержание

1 Введение

Тематическое моделирование — активно развивающаяся в последние годы область машинного обучения. Оно позволяет решать задачи тематического поиска, категоризации и кластеризации корпусов текстовых документов. Аналогичные задачи решаются для коллекций изображений и видеозаписей.

Тематическая модель определяет, к каким темам относится каждый документ, а также то, какие термины из словаря образуют ту или иную тему.

В работе [1] предлагается полувероятностный подход к тематическому моделированию — аддитивная регуляризация тематических моделей (APTM). Он позволяет записать любое количество дополнительных требований к тематической модели в виде взвешенной суммы критериев, добавляемых к основному функционалу логарифмированного правдоподобия. В [1] показано, что многие известные тематические модели допускают такое представление, то есть фактически являются лишь формой регуляризации. При этом, в отличие от стандартных задач машинного обучения — классификации и регрессии, в тематическом моделировании возникает огромное разнообразие регуляризаторов, направленных на улучшение модели и учёт дополнительной информации о текстовой коллекции. Задача тематического моделирования является по сути многокритериальной, и APTM позволяет выразить это непосредственным образом. Более распространённый в литературе байесовский подход основан на гипотезе о существовании адекватной вероятностной модели порождения текста. Эта гипотеза представляется избыточно сильной, так как далеко не все лингвистические требования и знания о естественном языке допускают вероятностую трактовку. Кроме того, техника байесовского вывода, как показывают работы последних лет, создаёт большое число технических трудностей при построении комбинированных моделей и попытках одновременного учёта большого числа разнородных требований.

Благодаря APTM появляется новая возможность — создать библиотеку из десятков различных регуляризаторов и строить решения прикладных задач путём обоснованного выбора подмножества регуляризаторов. В данной курсовой работе рассматривается архитектура и отдельные элементы библиотеки BigARTM, в которой в настоящее время реализуется данная концепция. Особое внимание уделяется мультипроцессорному и кластерному распараллеливанию, поскольку библиотека BigARTM изначально проектируется для тематического моделирования больших коллекций.

Работа имеет следующую структуру: в разделе 2 введены базовые обозначения и определения, необходимые для дальнейшего изложения; алгоритм обучения тематических моделей и идея регуляризации вводятся в разделе 3; в разделе 4 приводится краткий обзор существующих параллельных библиотек тематического моделирования; раздел 5 описывает общую архитектуру библиотеки и схему пользовательского взаимодействия с BigARTM; раздел 6 посвящён реализации и использованию регуляризаторов в библиотеке; в 7 разделе показаны эксперименты с регуляризаторами; раздел 8 предназначен для выводов и подведения итогов курсовой работы.

2 Терминология

Прежде всего рассмотрим некоторые базовые понятия и необходимые обозначения.

Вероятностная тематическая модель (BTM) описывает каждую тему дискретным рас-

пределением на множестве терминов, каждый документ — дискретным распределением на множестве тем. Предполагается, что коллекция документов — это последовательность терминов, выбранных случайно и независимо из смеси таких распределений, и ставится задача восстановления компонент смеси по выборке.

- D — коллекция текстовых документов.
- W — словарь коллекции текстов.
- T — множество тем.

Документы в коллекции можно представить в виде так называемого «мешка слов». В рамках этой концепции документ рассматривается как множество терминов из словаря и соответствующих им счётчиков частот встречаемости.

Замечание: «Мешок слов» на данный момент является основным способом представления коллекции, однако в последнее время всё большее развитие получают идеи хранения документа в виде последовательности слов. Порядок слов при этом становится важным и используется для улучшения качества обучения модели. BigARTM будет поддерживать оба способа представления.

После принятия гипотезы «мешка слов», коллекция представляется в виде матрицы $F_{W\times D}$, строки которой соответствуют терминам из словаря, а столбцы — документам коллекции. На пересечении строки и столбца находится оценка вероятности встретить данное слово в данном документе. Эта оценка является отношением числа раз, которое слово встретилось в документе, к общему числу слов в этом документе. Таким образом, столбцы матрицы F представляют собой распределения вероятностей.

При рассмотрении коллекции в виде пар (d, w), где w — номер термина, а d — номер документа, вводятся следующие счётчики частот:

- n_{dw} — число вхождений термина w в документ d;
- $n_d = \sum_{w\in W} n_{dw}$ — длина документа d в терминах;
- $n_w = \sum_{d\in D} n_{dw}$ — число вхожденией документа w во все документы коллекции;
- $n = \sum_{d\in D}\sum_{w\in d} n_{dw}$ — длина коллекции D в терминах;

Если же рассматривать коллекцию в виде троек (d, w, t), где d, w и t — номера соответствующих документа, термина и темы, то можно ввести такие счётчики:

- n_{dwt} — число троек, в которых термин w встретился в документе d и связан с темой t;
- $n_{dt} = \sum_{w\in W} n_{dwt}$ — число троек, в которых термин из документа d связан с темой t;
- $n_{wt} = \sum_{d\in D} n_{dwt}$ — число троек, в которых термин w связан с темой t;
- $n_t = \sum_{d\in D}\sum_{w\in d} n_{dwt}$ — число троек, связанных с темой t;

С использованием данных счётчиков можно ввести следующие частотные оценки вероятностей, связанных со скрытой переменной t:

- $\hat{p}(w|t) = \dfrac{n_{wt}}{n_t}, \quad \hat{p}(t|d) = \dfrac{n_{dt}}{n_d},$

Ставится задача разложения матрицы F в произведение двух матриц Φ и Θ меньшего размера, таких, что

- $\Phi = (\phi_{wt})_{W \times T}$, $\phi_{wt} = \hat{p}(w|t)$ — матрица «термины-темы»;
- $\Theta = (\theta_{td})_{T \times D}$, $\theta_{td} = \hat{p}(t|d)$ — матрица «темы-документы»;

Поставленная задача ($F \approx \Phi\Theta$) эквивалентна поиску матриц Φ и Θ, максимизирующих следующий функционал правдоподобия:

$$L(\Phi, \Theta) = \sum_{d \in D} \sum_{w \in d} n_{dw} \sum_{t \in T} \phi_{wt}\theta_{td} \to \max_{\Phi, \Theta} \tag{1}$$

3 Обучение тематических моделей

3.1 PLSA

Вероятностный латентный семантический анализ (PLSA) был предложен Т.Хофманном в [2].

Принимается гипотеза условной независимости, утверждающая, что вероятность появления термина в данном документе зависит только от темы этого термина и не зависит от документа. Вероятностная порождающая модель PLSA имеет следующий вид:

$$p(w|d) = \sum_{t \in T} p(w|t)p(t|d) \tag{2}$$

PLSA можно реализовать с помощью EM-алгоритма. Итерационный процесс состоит из двух шагов — E-шага (Expectation) и M-шага (Maximization). На E-шаге по текущим значениям ϕ_{wt} и θ_{td} с помощью формулы Байеса вычисляются условные вероятности $p(t|d, w)$:

$$H_{dwt} = p(t|d, w) = \frac{\phi_{wt}\theta_{td}}{\sum_{s \in T} \phi_{ws}\theta_{sd}}$$

На M-шаге по условным вероятностям H_{dwt} вычисляются новые приближения параметров ϕ_{wt} и θ_{td}. Используются указанные в предыдущем разделе формулы:

$$\phi_{wt} = \frac{n_{wt}}{n_t}, \quad \theta_{td} = \frac{n_{dt}}{n_d},$$

Однако данная версия алгоритма плохо подходит для задачи параллельной обработки бальших массивов данных по двум причинам:

- За время одного прохода по коллекции распределения терминов в темах (матрица Φ) успевают много раз сойтись, а распределния тем в документах проходят лишь одну итерацию.

- Трёхмерная вспогательная матрица H_{dwt} становится чрезвычайно большой.

Избавиться от этих проблем позволяет т.н. пакетный онлайновый EM-алгоритм. Пакетным он называется потому, что вся коллекция делится на блоки документов, обрабатываемых независимо. Онлайновость означает потоковую обработку корпуса, когда модель дообучается, получая на вход всё новые документы.

Помимо описанных преимуществ, данная модификация позволят также не хранить в явном виде матрицу Θ. Вероятности θ_{td} не нужны по окончании обработки документа d, поэтому матрицу $(\theta_{td})_{T\times D}$ можно заменить вектором $(\theta_t)_T$.

Замечание: Работа онлайнового алгоритма отличается для больших и малых коллекций. Для большой коллекции достаточно одного прохода по всем документам, в то время как маленькие требуют многократного прохода. В случаях малых и динамических коллекций (т.е.меняющих со временм свою тематику) значимость пакетов убывает по мере поступления новых, поэтому необходимо ввести параметр $\rho_j \in (0, 1]$, отвечающий за скорость «забывания» старых оценок:

$$n_{wt} := \rho_j n_{wt} + \tilde{n}_{wt}$$

где $\tilde{n}_{wt}$ — новые полученные счётчики.

Алгоритм Online Batch PLSA реализует все описанные идеи. Он выглядит следующим образом:

Online Batch PLSA

инициализировать ϕ_{wt} для всех $w \in W$ и $t \in T$;
$n_{wt} := 0, n_t := 0$ для всех $w \in W$ и $t \in T$;
для всех пакетов D_j, j = 1,...,J **выполнять**
  повторять
    $\tilde{n}_{wt} := 0, \tilde{n}_t := 0$ для всех $w \in W$ и $t \in T$;
    для всех $d \in D_j$ **выполнять**
      инициализировать θ_{td} для всех $t \in T$;
      повторять
        $Z_w := \sum_{t\in T} \phi_{wt}\theta_{td}$ для всех $w \in d$;
        $\theta_{td} := \dfrac{1}{n_d}\sum_{w\in d} n_{dw}\phi_{wt}\theta_{td}/Z_w$ для всех $t \in T$;
      пока θ_d не сойдётся;
      увеличить $\tilde{n}_{wt}, \tilde{n}_t$ на $n_{dw}\phi_{wt}\theta_{td}/Z_w$ для всех $w \in W$ и $t \in T$;
    $\phi_{wt} := \dfrac{\rho_j n_{wt} + \tilde{n}_{wt}}{\rho_j n_t + \tilde{n}_t}$ для всех $w \in W$ и $t \in T$ таких, что $\tilde{n}_{wt} > 0$;
  пока Φ не сойдутся;
  $n_{wt} := \rho_j n_{wt} + \tilde{n}_{wt}$ для всех $w \in W$ и $t \in T$;
  $n_t := \rho_j n_t + \tilde{n}_t$ для всех $t \in T$;

Такой алгоритм хорошо параллелится. Θ выводится на нодах, Φ — на мастере.

3.2 Аддитивная регуляризация

Неоднозначность матричного разложения $F \approx \Theta\Phi$ даёт свободу выбора матриц из правой части равенства, позволяя наложить на тематическую модель дополнительные требования. Модифицируем максимизируемый функционал 1:

$$L(\Phi, \Theta) + R(\Phi, \Theta) \to \max_{\Phi,\Theta} \tag{3}$$

$$R(\Phi, \Theta) = \sum_{i=1}^{n} \tau_i R_i(\Phi, \Theta) \tag{4}$$

где $R_i(\Phi, \Theta)$ — дополнительные требования к модели, τ_i — неотрицательные коэффициенты регуляризации, выполнены условия неотрицательности и нормировки столбцов матриц Φ и Θ.

Решение этой задачи приводит к обощению формул M-шага в EM-алгоритме:

$$\phi_{wt} = \frac{\left(n_{wt} + \phi_{wt}\dfrac{\partial R}{\partial \phi_{wt}}(\Phi, \Theta)\right)_+}{\sum_{u \in W}\left(n_{ut} + \phi_{ut}\dfrac{\partial R}{\partial \phi_{ut}}(\Phi, \Theta)\right)_+}, \quad \theta_{td} = \frac{\left(n_{dt} + \theta_{td}\dfrac{\partial R}{\partial \theta_{td}}(\Phi, \Theta)\right)_+}{\sum_{s \in T}\left(n_{ds} + \theta_{sd}\dfrac{\partial R}{\partial \theta_{sd}}(\Phi, \Theta)\right)_+} \tag{5}$$

n_{wt} и n_{dt} определяются аналогично из формул предыдущего раздела.

Таким образом, суть добавления регуляризаторов — в простом изменении формул M-шага.

Замечание: Использование регуляризаторов требует аккуратного выстраивания т.н. траектории регуляризации. Этот процесс включает в себя настройку параметров, определение времени подключения/отключения того или иного регуляризатора, используемого в модели и т.п.

4 Существующие реализации параллельных алгоритмов

В данном разделе кратко рассматриваются некоторые известные параллельные алгоритмы тематического моделирования. Все они основаны на модели LDA, имеют разные технические и алгоритмические детали. Рассмотрим их основные преимущества и недостатки, после чего опишем на базе полученных выводов ключевые требования к BigARTM.

4.1 Краткий обзор

4.1.1 AD-LDA

Данный алгоритм был предложен в работе [4]. В AD-LDA был реализован только межпроцессорный параллелизм, возможность работы на кластере не обсуждалась.

Представленная архитектура предполагает размещение документов в случайном порядке на каждом из участвующих в вычислениях процессоров. Все процессоры одновременно производят сэмплирование, получая счётчики матрицы Θ. Этот процесс происходит до тех пор, пока последний процессор не закончит работу. После производится шаг синхронизации, на котором счётчики со всех процессоров сливаются вместе. Затем происходит обновление глобальной матрицы Φ. Новая Φ рассылается каждому процессору, и начинается следующая итерация симплирования. И так до тех пор, пока не выполнится заданный критерий останова.

Данная работа является одной из первых, в которых была осознана возможность параллелизации процеса обучения модели, и в этом её основное достоинство. Но сама реализация имеет ряд серьёзных недостатков:

1. Скорость работы алгоритма определяется скоростью самого медленного процессора.
2. Нагрузка на сеть неравномерна — высокая во время синхронизации и почти нулевая в процессе сэмплирования.
3. Система обладает низкой отказоустойчивостью.
4. Наличие отдельной матрицы Φ у каждого процессора приводит к большим затратам памяти.

Как видно, большинство недостатков возникают в виде следствия наличия отдельного шага синхронизации.

4.1.2 PLDA

PLDA был описан в статье [5]. Алгоритм представляет собой модификацию AD-LDA, в которой параллелизм реализован в двух вариантах: с помощью MPI и Map-Reduce (при этом работа опять же распределяется между процессорами, кластерная реализация отсутствует).

Архитектура не претерпела больших изменений. Существенной модификацией является использование контрольных точек — откачки данных после очередной итерации сэмплирования на жёсткий диск. В случае сбоя имеется возможность восстановить данные и продолжить вычисления. В MPI-реализации это действительно работает хорошо. В Map-Reduce версии реализация контрольных точек требует больших затрат памяти. Авторы сами рекомендуют использовать MPI.

Собственно, помимо низкой отказоустойчивости, PLDA сохранил все недостатки AD-LDA.

4.1.3 Y!LDA

Следующим шагом для параллельных алгоритмов обучения тематических стал Y!LDA, описанный в [6]. Авторы рассмотрели недостатки AD-LDA и предложили свой вариант параллелизма, теперь уже как в рамках одной машины, так и на кластере. Основным достижением стало избавление от выделенного шага синхронизации. Авторы отметили его основные недостатки (плохая отказоустойчивость, неравномерная нагрузка на сеть и

ожидание медленных процессоров) и предложили способ их разрешения, который будет рассмотрен далее.

Коллекция делится между машинами. В свою очередь, в рамках каждой ноды документы распределяются по процессорам. Все процессоры одного компьютера имеют общую матрицу Φ, что существенно снижает затраты памяти. На каждой ноде имеется т.н. поток синхронизации, к которому ядра обращаются независимо по мере завершения обработки очередной порции документов.

Глобальная матрица Φ хранится в `memcached` [1] . Работа кластера организована с использованием т.н. «архитектуры классной доски». Суть её состоит в том, что каждый компьютер в сети обращается к глобальной матрице Φ независимо от других, обновляя полученные им счётчики. Порцией обновления при этом является один термин. Кластерный параллелизм был реализован в Hadoop.

4.1.4 PLDA+

В публикации [7] была предпринята попытка модифицировать PLDA. Авторы сделали алгоритм параллельным в кластерном смысле. Для того, чтобы побороть проблемы, связанные с шагом синхронизации, было решено использовать конвейерную архитектуру. Для этого всё множество процессоров делится на две группы — вычислители и коммутаторы. Первая группа производит сэмплирование (собственно расчёт модели), вторая отвечает за своевременную доставку данных. Вкупе с равномерным распределением коллекции в виде блоков по процессорам-вычислителям, а также наличием механизма приоритетов, сглаживающем узкие места во время счёта, это даёт, согласно экспериментам, хорошие результаты. Тем не менее, один существенный недостаток у данной реализации имеется — значимая часть вычислительных ресурсов тратится не на счёт, а на своевременную доставку данных.

4.2 BigARTM

Из всего описанного были сделаны следующие выводы:

1. Алгоритм обязательно должен быть асинхронным, что позволит избежать недостатков AD-LDA.

2. Матрица Φ должна быть локальной в рамках компьютера, а не ядра, что приведёт к существенной экономии оперативной памяти.

3. Использование `memcached` является хорошим решением для хранения глобальной матрицы Φ [2] .

[1] memcached — связующее программное обеспечение, реализующее сервис кэширования данных в оперативной памяти на основе парадигмы хеш-таблицы.

[2] К сожалению, сервис memcached предназначен для использования только под Linux. Тем не менее, это не является большой проблемой, поскольку большинство кластеров работает именно под Unix-подобными ОС и кластерный параллелизм BigARTM будет ориентирован именно под них. Windows-версия библиотеки, по сути, предназначена для использования на локальной машине. Хотя, возможно, к релизу для Windows-версии будет написан сервис, осуществляющий функции memcached, как это уже сделано в рамках многопроцессорного параллелизма.

4. Коллекцию следует разделить на пакеты документов, обрабатываемых независимо.

Все эти идеи положены в основу BigARTM. Они либо уже реализованы, либо будут реализованы к моменту релиза библиотеки. Таким образом, BigARTM архитектурно похожа на Y!LDA, за исключением использованного фреймворка для организации кластерного параллелизма (MPI вместо Hadoop).

5 Библиотека BigARTM

BigARTM — библиотека тематического моделирования, реализующая концепцию аддитивной регуляризации. На данный момент библиотека поддерживает многопроцессорный параллелизм. Основным алгоритмом является описанный в 3.1 Online Batch PLSA.

Protocol buffers. [3] Рассмотрим кратко эту технологию, поскольку в BigARTM она используется очень интенсивно. Protocol Buffers позволяет описывать proto-сообщения на псевдоязыке, которые затем можно преобразовать специальным компилятором (protoc) в структуры данных со всеми необходимыми методами на C++, Python и Java. Ключевой особенностью является возможность обмена сообщениями между этими языками программирования посредством механизма serializer/deserializer, переводящего сообщения в байт-массив и обратно. Данное решение является максимально переносимым и унифицированным.

Все конфигурации библиотеки, такие, как настройки тематической модели, параметры регуляризаторов и т.п., определяются и передаются только посредством proto-сообщений. Результирующая модель также описывается соответствующим сообщением.

Замечание: Все необходимые для работы с библиотекой сообщения будут описаны при дальнейшем изложении.

Далее будет рассмотрено краткое описание библиотеки, основные шаги по настройке и использованию существующей версии BigARTM, а также список нововведений, которые будут в неё внесены в будущем (либо уже внесены, но требуют доработки).

5.1 Краткое описание

5.1.1 Представление коллекции

Коллекция представляется в виде «мешка слов» (Bag of words). Поскольку библиотека предназначена для обработки больших массивов текстовой информации, она является онлайновой. Механизм потоковой загрузки и обработки информации реализуется разделением коллекции на пакеты (`Batch`), которые обрабатываются по очереди. Каждый `Batch` имеет свой словарь, который может быть как локальным (что предпочтительнее), так и совпадающим со всем словарём коллекции.

Приведём здесь соответствующее proto-сообщение:

[3] Подробнее о том, что такое Google Protocol Buffers и как с ними работать, можно прочесть в [3]

```
message Batch {
  repeated string token = 1;
  repeated Item item = 2;
}
```

Первое поле — набор терминов (словарь), второе — набор документов (см. 5.1.2).

Замечание: В силу внутренних особенностей работы BigARTM, оптимальное число `Batches` — в 4-5 раз больше, чем число используемых процессоров. Этот параметр не повлиет на качество результирующей модели, но скажется на производительности.

5.1.2 Представление документов

В BigARTM реализована гибкая концепция представления данных. Каждый документ является экземпляром класса `Item`. Этот объект, помимо самого документа (хранимого в виде последовательности терминов и их счётчиков), может содержать произвольные метаданные, связанные с ним. К таким данным относятся информация об авторе, дате публикации, ссылках на документ и из него, тегах и т.п. Всё это может оказаться крайне полезным при использовании тех или иных регуляризаторов.

Рассмотрим proto-сообщение для `Item`

```
message Item {
  optional int32 id = 1;
  repeated Field field = 2;
}
```

Здесь `id` — идентификатор документа, второе поле — набор всевозможных объектов в тексте. Объектом может быть сам текст документа, строка с авторами, список тегов и т.п. Работа механизма становится ясной, если взглянуть на определение сообщения для `Field`

```
message Field {
  optional string field_name = 1 [default = "@body"];
  repeated int32 token_id = 2;
  repeated int32 token_count = 3;
}
```

Первое поле сообщает о том, что это за объект (по-умолчанию — основной текст документа). Второе поле содержит список идентификаторов терминов, встречающихся в тексте этого объекта. В третьем находятся соответствующие этим терминам счётчики встречаемости в данном документе.

5.1.3 Входные и выходные данные

На вход функциям библиотеки могут подаваться как исходные тексты, так и коллекция в виде пакетов. Это крайне удачное решение, поскольку, во-первых, коллекция в пакетном виде занимает более чем в два раза меньше памяти [4] , чем в исходном текстовом, во-

[4]Никакой архивации при этом не производится. Каждый Batch представляет собой набор Item в виде сериализованных proto-сообщений. Это было сделано из соображений удобства использования, а объём уменьшается в виде побочного эффекта.

вторых, библиотеке не придётся производить процедуру преобразования, что сэкономит время. На выходе пользователь получает собственно тематическую модель (матрицу Φ).

5.1.4 Архитектура библиотеки

Для начала рассмотрим на концептуальном уровне параллелизм в рамках одной машины. На каждый процессор, участвующий в работе, создаётся экземпляр класса `Processor`. Объекты этого класса отвечают за вывод столбцов матрицы Θ, каждый из них работает независимо от других и обрабатывает свой пакет документов. Для процессоров имеется т.н. очередь процессоров, откуда `Processor` извлекает очередной блок документов, закончив обработку предыдущего. Кроме того, существует т.н. очередь слияния, куда все процессоры отправляют полученные результаты. Сама матрица Θ при этом в явном виде нигде не хранится. Важно заметить, что у процессоров нет никакого «шага синхронизации», вся обработка и отправка результатов осуществлется асинхронно.

В рамках компьютера создаётся единственный объект класса `Merger`. Он играет роль мастера при межпроцессорном параллелизме. Его задача — извлекать из очереди слияния данные, полученные процессорами, и обновлять матрицу Φ. `Merger` в явном виде хранит две версии матрицы Φ — «активную» и «базовую». К первой обращаются во время своей работы процессоры, вторую `Merger` может безопасно обновлять. `Merger` определяет, когда нужно переключиться на новую «активную» матрицу. Тем не менее, все процессоры, начавшие обработку очередного пакета со старой матрицей, с ней и будут работать. Доступ к новой матрице они получат, начав обработку нового блока документов. Все операции с матрицами Φ не затратные — удаления, замены, отсылка данных производятся посредством работы с указателями. Все используемые данные хранятся в оперативной памяти, новые пакеты документов подгружаются с жёсткого диска.

Замечание: На самом деле, нет строгого запрета задавать функциям библиотеки большее число процессоров, чем их фактически существует. Однако это увеличит нагрузку на `Merger`, что приведёт к снижению эффективности работы. Рекомендация — указывать число используемых процессоров не превосходящим числа физических процессоров.

Кластерный параллелизм находится в данный момент на стадии разработки. Ноды будут работать по принципу, описанному выше. На мастере будет создан экземпляр класса `NetworkMerger`, отвечающий за слияние данных, полученных на локальных `Merger`, а также за предоставление им актуальной информации о матрице Φ.

5.2 Установка

Для установки библиотеки необходимо выполнить следующую последовательность действий:

1. Скачать по адресу `http://miru.hk/archive/ZeroMQ-4.0.3~miru1.0-x86.exe` библиотеку ZeroMQ, распаковать её, и присвоить системной переменной `ZEROMQ32\_ROOT` путь к корневой директории ZeroMQ (если переменная не существует, необходимо создать её).

2. Установить Python 2.7 (x32)

3. Добавить корневую папку Python в переменную окружения `PATH`.

4. Следовать инструкциям, описанным в файле

   `BigARTM_ROOT_DIRECTORY\protobuf\python\README.txt`

5.3 Работа с библиотекой

Замечание: На данный момент единственным поддерживаемым библиотекой языком является Python, все выкладки будут производится на нём.

Замечание: Здесь и далее под внешней итерацией работы алгоритма понимается один проход по всей коллекции, а под внутренней — один проход по одному документу (`Item`).

Запуск обучения. Основным классом, обеспечивающем пользовательский API, является `MasterComponent`. Объект данного класса содержит в себе все созданные тематические модели и регуляризаторы, через него производится управление процессом обучения.

Для того, чтобы запустить обучение тематической модели, требуется пошаговое выполнение следующей инструкции:

1. Ядро библиотеки представляет собой скомпилированный модуль `artm.dll`. Прежде всего, требуется подключить его. Чтобы это сделать, необходимо поместить в программу следующие строки:

   ```
   os.environ['PATH'] = ';'.join([address +
     '\\Win32\\Release', os.environ['PATH']])
   library = ArtmLibrary(address + '\\artm.dll')
   ```

   Здесь `address` — местоположение модуля библиотеки.

2. Необходимо создать объект класса `MasterComponent`. Для этого опишем соответствующее proto-сообщение, которое (в базовой конфигурации) требует заполнения следующих полей:

   ```
   optional int32 processors_count = 2 [default = 1];
   optional string disk_path = 3;
   ```

   Первое поле представляет собой число процессоров, в рамках которых будет производится распараллеливание алгоритма. Второе поле описывает путь к папке, в которую будут сохранятся данные, преобразованные в `Batches` (кроме того, именно в этой папке библиотека будет искать пакеты для своей работы). Создание и описание этого сообщения может иметь следующий вид:

   ```
   master_config = messages_pb2.MasterComponentConfig()
   master_config.processors_count = 1
   master_config.disk_path = 'disk_path'
   ```

   После того, как конфигурационное сообщение сформировано, можно создать сам объект `MasterComponent`:

   ```
   master_component = library.CreateMasterComponent(master_config)
   ```

3. Следующим шагом требуется загрузить коллекцию и словарь для неё. Данное действие является техническим, в качестве базового примера можно использовать считывание, описанное в `python_client.py`.

4. Теперь в `master_component` нужно добавить тематические модели для обучения. Допустим, что требуется обучить одну модель. Для её создания необходимо заполнить соответствующее proto-сообщение. Оно будет выглядеть так:

```
optional string name = 1 [default = ""];
optional int32 topics_count = 2 [default = 32];
optional int32 inner_iterations_count = 4 [default = 10];
repeated Score score = 7;
repeated string regularizer_name = 10;
repeated double regularizer_tau = 11;
```

   Параметры имеют следующие назначения: `name` — имя тематической модели; `topic_counts` — число тем, которые будет искать в коллекции данная модель; третий параметр назначает модели число внутренних итераций; в поле `score` указываются функционалы качества, которые наобходимо рассчитывать для данной модели в ходе её работы [5] . Последние два поля будут рассмотрены в разделе 6. Создать конфигурацию модели можно так:

```
model_config = messages_pb2.ModelConfig()
model_config.topics_count = 20
model_config.inner_iterations_count = 10
score_ = model_config.score.add()
score_.type = 0
```

   `score_.type = 0` соответствует добавлению модель требование подсчёта перплексии на каждой итерации. Теперь можно создать саму модель, отнеся её к созданному ранее объекту `MasterComponent`:

```
model = master_component.CreateModel(master_component, model_config)
```

5. После того, как модель была создана, необходимо запустить работу алгоритма [6] . Для этого нужно описать цикл по числу внешних итераций (это число определяется пользователем), внутри которого должны быть такие строки кода:

```
master_component.InvokeIteration(1)
master_component.WaitIdle();
```

   Первая строка производит вызов одной внешней итерации работы алгоритма, вторая — ожидание завершения выполнения это итерации.

   Этого достаточно для запуска алгоритма, однако в большинстве случаев требуется контролировать его работу (в т.ч. и следить за перплексией на данной итерации). Для

[5] На данный момент в библиотеке реализована только перплексия.

[6] Вообще говоря, прежде, чем начинать счёт, в модель стоит добавить регуляризаторы. О том, как это сделать, подробно написано в соответствующем разделе.

этого следует выгрузить посчитанную на данный момент модель и просмотреть её параметры. Выгрузить модель можно, добавив ещё одну строку под предыдущими:

```
topic_model = master_component.GetTopicModel(model)
```

Как было указано ранее, сама модель описывается proto-сообщением. Это сообщение имеет такой вид:

```
optional string name = 1 [default = ""];
optional int32 topics_count = 2;
optional int32 items_processed = 3;
repeated string token = 4;
repeated FloatArray token_weights = 5;
optional DoubleArray scores = 6;
```

Первые два поля были описаны ранее и имеют тот же смысл. Третье содержит число обработанных на данный момент документов (учитываются все проходы по каждому документу, включая повторные). В четвёртом поле лежит словарь для данной модели. В пятом — веса каждого термина, полученные в результате работы алгоритма (матрица Ф). Последнее поле содержит значения счётчиков на данной итерации (в описываемом примере — значение перплексии). Данную информацию можно извлекать после каждой итерации и использовать для оценивания качества обучения.

Выгруженная после последней внешней итерации тематическая модель является финальным результатом работы алгоритма.

6. В случае необходимости перенастройки `master_component` или `model` используется функция `Reconfigure()`:

   ```
   master_component.Reconfigure(new_master_config)
   model.Reconfigure(new_model_config)
   ```

7. После окончания работы модели удалять её не нужно. Все модели будут автоматически удалены при вызове деструктора `MasterComponent`.В случае, когда необходимо прекратить счёт модели до завершения работы с библиотекой, можно деактивировать её вызовом:

   ```
   model.Disabled()
   ```

   Если нужно снова активировать — описать вызов:

   ```
   model.Enabled()
   ```

Замечание: Базовых типов `DoubleArray` и `FloatArray` в protocol buffers нет, это пользовательские тип, эквивалентные вещественному массиву (работать с ними нужно по тем же правилам, что и остальными сообщениями — через создаваемый protocol buffers интерфейс).

Замечание: Все функции типа `Reconfigure()` являются не до конца тестированными, поэтому должны использоваться с осторожностью.

Замечание: Пример пользовательского приложения над BigARTM можно найти в файле `BigARTM_ROOT_DIRECTORY\python_client`. Выдержки кода, указанные в вышеописанной инструкции, заимствованы оттуда.

5.4 Планируемые нововведения

Ниже приведён список различных модификаций, которые появятся в release-версии библиотеки:

- Кластерный параллелизм.
- Возможность работы в Linux.
- Интерфейсы на C++ и Java.
- Коллекция регуляризаторов.
- Поддержка 64-битной архитектуры.
- Усовершенствованный механизм хранения данных, необходимых для работы регуляризаторов.
- Коллекция функционалов качества.
- Хранение матриц Φ и Θ в разреженном виде.
- Возможность классификации новых документов построенной тематической моделью.

6 Регуляризаторы в BigARTM

Ключевым отличием BigARTM от других библиотек алгоритмов тематического моделирования является наличие регуляризаторов. Задача данного раздела — описать API, предоставленный библиотекой для работы с существующими реализациями регуляризаторов, а также пояснить схему добавления новых.

6.1 Краткое описание механизма регуляризации в BigARTM

Поскольку части матрицы Θ вычисляются независимо на разных процессорах, регуляризация Θ производится в каждом процессоре на всех енних итерациях. Объектом регуляризации является столбец матрицы (т.е. документ). Управлять процессом регуляризации Θ можно только между внешними итерациями. С матрицей Φ ситуация иная. Регуляризация происходит в `Merger`, после того, как все процессоры на данной внешней итерации отработали, и новая матрица Φ сформирована. Регуляризация в логическом смысле опять работает со столбцом (т.е. с темой), но технически регуляризаторы вызываются для каждого элемента матрицы. При этом нормировочная константа, необходимая для выполнения M-шага 3 алгоритма, автоматически поправляется. Φ регуляризуется только на тех внешних итерациях, на которых это указал пользователь вызовом соответствующей функции (см. ниже).

6.2 Существующие регуляризаторы

6.2.1 Регуляризатор Дирихле

Рассмотрим популярную на сегодняшний день модель обучения LDA (latent Dirichlet allocation). Она основана на разложении 2 при дополнительном предположении, что векторы документов $\theta_d = (\theta_{td} \in \mathbf{R}^{|T|})$ и векторы тем $\phi_t = (\phi_{wt} \in \mathbf{R}^{|W|})$ порождаются распределениями Дирихле с параметрами $\alpha \in \mathbf{R}^{|T|}$ и $\beta \in \mathbf{R}^{|W|}$ соответственно. В работе [8] показано, что отличие LDA от PLSA — в сглаживании ϕ_{wt} и θ_{td}:

$$\phi_{wt} = \frac{\hat{n}_{wt} + \beta_w}{\hat{n}_t + \sum_w \beta_w}, \quad \theta_{td} = \frac{\hat{n}_{td} + \alpha_t}{\hat{n}_d + \sum_t \alpha_t}, \tag{6}$$

T.e. LDA — это PLSA со встроенным регуляризатором сглаживания (здесь и далее — регуляризатор Дирихле). Сглаживание может быть полезным для некоторых столбцов матриц, но в общем случае оно противоречит тому, что матрицы Φ и Θ должны быть сильно разреженными.

6.2.2 Регуляризатор сглаживания/разреживания

[7]

Данный регуляризатор представляет собой модификацию сглаживающего регуляризатора Дирихле. Несмотря на простоту, уже данный регуляризатор решает задачи более сложные, чем регуляризатор Дирихле.

Разделим множество тем T на предметные S и фоновые B. Зададим вектор-столбцы параметров таким образом, чтобы в обеих матрицах Φ и Θ темы из S одинаково разреживались, а темы из B — сглаживались каждая собственным образом. Это приводит к тому, что

- каждая тема из S приобретает некоторое выраженное ядро терминов, отличающее её от остальных тем;
- каждая фоновая тема из B сглаживается, причём вариация параметров приводит к тому, что разные темы выполняют различные задачи (выделение стоп-слов, общей лексики коллекции и т.п.).

6.3 Реализованные в BigARTM регуляризаторы

Все регуляризаторы описываются своими proto-сообщениями, содержащими необходимые данные и параметры. Конфигурационное сообщение для регуляризатора матрицы Θ должно содержать по одному набору параметров для каждой внутренней итерации. Сообщение для регуляризатора матрицы Φ наполняется одним набором параметров для одной внешней итерации. По истечении внешней итерации все конфигурации могут быть обновлены.

[7] Подробное описание и лингвистические обоснования этого регуляризатора можно найти в [8]

6.3.1 Механизм словарей

В качестве параметра регуляризатора могут использоваться т.н. словари — структура данных, proto-сообщение для которых имеет следующий вид:

```
message DictionaryConfig {
  required string name = 1;
  repeated DictionaryEntry entry = 2;
}
message DictionaryEntry {
  required string key_token = 1;
  optional float value = 2;
  repeated string value_tokens = 3;
  optional FloatArray values = 4;
}
```

Первое поле сообщения `DictionaryConfig` — название словаря, второе — список содержимого. Каждый элемент словаря (`DictionaryEntry`) имеет четыре поля. `key_token` — собственно слово, `value` — какое-то числовое значение, соответствующее этому слову (например, частота встречаемости данного слова в коллекции). `value_tokens` — это некие термины, которые соответствуют данному слову (это могут быть переводы данного слова на другой язык и т.п.). Последнее поле — это набор чисел, который может использоваться для хранения значений, необходимых для использования данного словаря. Словари являются крайне гибким способом задания параметров регуляризатора, поскольку могут содержать в себе самую разнообразную информацию.

Механизм устроен следующим образом. Словари, используемые в определённых регуляризаторах, должны иметь нужную структуру (назначения полей в `DictionaryEntry`), которая документируется на этапе написания регуляризатора его авторами. Для того, чтобы использовать словари, необходимо создать конфигурацию (заполнить описанное выше сообщение данными нужного вида) — это можно сделать как во время работы библиотеки, так и до с помощью сторонней программы. После создания [8] сообщения, его необходимо передать `MasterComponent`, который создаст объект словаря. Все словари хранятся в виде списка в `MasterComponent`, ключом при поиске нужного словаря является его имя. Регуляризаторы, которым для работы могут понадобиться словари, имеют в своём конфигурационном сообщении поле `dictionary_name`, которое содержит имена необходимых словарей. При создании или реконфигурации регуляризатор получает указатели на нужные словари.

6.3.2 Реализации регуляризаторов

На данный момент в BigARTM реализовано по одному базовому регуляризатору для каждой из матриц Φ и Θ, информация о которых приведена ниже:

[8] Подробно процесс создания будет рассмотрен в секции «Подключение регуляризаторов».

Имя регуляризатора	**protobuf-сообщение**

Регуляризатор сглаживания/ разреживания для Θ	`message SmoothSparseThetaConfig {`
	`  required int32 background_topics_count = 1`
	`  repeated DoubleArray alpha = 2;`
	`}`

Регуляризатор сглаживания/ разреживания для Φ	`message SmoothSparsePhiConfig {`
	`  required int32 background_topics_count = 1;`
	`  optional string dictionary_name = 2;`
	`}`

Замечание: В данной реализации регуляризатора фоновыми считаются $|B|$ тем с конца.

Для регуляризатора Θ каждый набор параметров состоит из вектора длиной в число тем (`alpha`). Кроме того, имеется параметр числа фоновых тем `background_topics_count`, который нужно указать. `alpha` должен содержать отрицательные числа для тем из S, и положительные — для тем из B.

Регуляризатор для Φ так же требует в явном виде указать число фоновых тем (поле `background_topics_count`) — это нужно для проверки корректности параметров. Во втором поле предполагается наличие имени словаря, который нужен этому регуляризатору. Поле `value` в данном словаре — это частота данного термина в коллекции (может быть посчитана во время первого прохода по коллекции), последние два поля для работы не нужны и могут оставаться пустыми. Встречая в документе термин, регуляризатор будет обращаться к словарю за значением поля `value` для этого термина. Если термина в словаре нет, либо поле `value` для него не определено, то регуляризация для данного слова производится не будет. В противном случае полученная частота будет являться элементом описанного выше выктора парметров β, соответствующим данному термину.

Замечание: Выгода от использования словарей очевидна — в ситуации, когда используется несколько регуляризаторов, которым требуется один и тот же набор данных, достаточно содать один словарь. Он будет использоваться всеми заинтересованными регудяризаторами и храниться в памяти в одном экземпляре. Кроме того, использование статических векторов β (т.е. способа хранения, используемого в конфигурации `SmoothSparseTheta`) приводит к зависимости регуляризатора от номеров терминов, который, вообще говоря, меняется при добавлении/удалении слов в коллекции. Словари же дают возможность непосредственной работы с терминами. Кроме того, механизм словарей — крайне удобный инструмент при реализации мультиязычных регуляризаторов.

Замечание: Регуляризатор сглаживания/разреживания превращается в регуляризатор Дирихле очевидным образом: достаточно задать нужным образом все элементы `alpha` для Θ и словарь для Φ и присвоить ноль переменным `background_topics_count`.

Значения параметров по-умолчанию. В случае, если вектор параметров α для регуляризатора Θ не указан, либо отсутствует имя словаря для регуляризатора Φ, по-умолчанию используются единичные значения. Сделано это, в первую очередь, для удобства — если все значения будут одинаковыми, достаточно регулировать их соответствующим коэффициентом регуляризации τ (4). Следует обратить внимание на то, что в случае, когда число наборов параметров α меньше, чем число внутренних итераций, все имеющиеся α будут использованы на первых итерациях. На всех оставшихся будут использоваться единичные векторы.

6.4 Подключение регуляризаторов

Теперь, на примере `SmoothSparseTheta` и `SmoothSparsePhi`, будет рассмотрен поэтапно процесс включения регуляризатора в тематическую модель. Предполагается, что переменная `topic_counts` содержит общее число тем, выделяемых данной моделью, а `background_topics_count` — число тем, объявленных фоновыми. В случае, когда описание будет одинаковым для регуляризаторов Φ и Θ, будут показаны действия с `SmoothSparseTheta`.

1. Прежде всего следует создать конфигурационное сообщение для регуляризатора. Оно может быть описано так:

```
regularizer_config_theta = messages_pb2.SmoothSparseThetaConfig()
regularizer_config_theta.background_topics_count =
  background_topics_count
for i in range(0, inner_iterations_count):
  alpha_ref = regularizer_config_theta.alpha.add()
  for j in range(0, topics_count - background_topics_count):
    alpha_ref.value.append(-0.5)
  for j in range(0, background_topics_count):
    alpha_ref.value.append(0.5)

regularizer_config_phi = messages_pb2.SmoothSparsePhiConfig()
regularizer_config_phi.background_topics_count =
  background_topics_count
regularizer_config_phi.dictionary_name = 'phi_dictionary'
```

2. Для создания регуляризатора используется функция `CreateRegularizer()`. Она имеет следующий вид:

```
MasterComponentObject.CreateRegularizer(name, type, config)
```

   Здесь первое и второе поля — имя и тип создаваемого регуляризатора соответственно. Имя — это строка, тип — число, определённое при создании регуляризатора (описанным выше регуляризаторам сглаживания/разреживания соответствуют типы 2 и 3). Третье поле — это описанная выше конфигурация самого регулzризатора. В рассматриваемом примере вызов может быть таким:

```
regularizer_theta = master_component.CreateRegularizer(
  'regularizer_theta', 2, regularizer_config_theta)
```

Добавляемый регуляризатор сохранён в данном `MasterComponent`, все модели, ассоциированные с этим `MasterComponent`, могут использовать его.

Вспомним, что регуляризатору Φ для работы требуется словарь. Процесс получения этого словаря оставляется на усмотрение пользователя (пример описан в `python_client.py`). Допустим, что конфигурационное сообщение `dictionary_config` словаря получено (поле `name = 'phi_dictionary'`). Создание словаря в $MasterComponent$ производится следующим образом:

```
dictionary = master_component.CreateDictionary(dictionary_config)
```

Теперь словарь хранится а `master_component` и все регуляризаторы, в конфигурации которых указано название этого словаря, смогут его использовать.

3. Теперь требуется сообщить нужной тематической модели о том, что при её обучении должен использоваться `regularizer_theta`. Для этого требуется модификации шага 4 в алгоритме работы с библиотекой. Перед созданием модели её конфигурация дополняется ещё двумя строками:

```
model_config.regularizer_name.append('regularizer_theta')
model_config.regularizer_tau.append(1)
```

Таким образом модель «узнаёт» имя регуляризатора, который нужно будет использовать, а также его коэффициент регуляризации. Разные модели могут иметь разные списки используемых регуляризаторов. При этом все эти регуляризаторы, как уже было отмечено, будут храниться в активном экземпляре `MasterComponent`.

4. Регуляризация Θ после добавления регуляризатора в модель будет производиться автоматически на каждой внутренней итерации. В случае, когда на данной внешней итерации нужно произвести регуляризацию Φ, необходимо в шаге 5 алгоритма использования библиотеки после строки

```
topic_model = master_component.GetTopicModel(model)
```

сразу вставить строку

```
topic_model.InvokePhiRegularizers();
```

Если для регуляризации требуется обновить конфигурации регуляризаторов, делать это нужно между выгрузкой модели и вызовом `InvokePhiRegularizers();`, поскольку текущие данные модели могут быть нужны для настройки параметров регуляризации. В случае, если данная внешняя итерация является последней, для учёта регуляризации требуется ещё раз выгрузить модель уже после вызова регуляризации и полученный результат считать финальным.

Стоит обратить внимание на то, что при регуляризации будут вызваны все регуляризаторы матрицы Φ, ассоциированные с данной моделью.

5. Часто возникает потребность заменить параметры существующего регуляризатора. Для этого необходимо описать новое конфигурационное сообщение, после чего вызвать функцию перенастройки:

```
regularizer_theta.Reconfigure(new_regularizer_config_theta)
```

Аналогичным образом заменяется соответствующий какому-либо регуляризатору параметр τ, а также производится обновление словарей.

6. Для удаления регуляризатора из модели достаточно удалить его имя из списка `model_config.regularizer_name`, а также удалить коэффициент τ из списка `model_config.regularizer_tau`, после чего реконфигурировать модель обновлённой `model_config`. Сам `regularizer_theta` будет удалён автоматически при завершении работы `MasterComponent`.

6.5 Создание нового регуляризатора

[9]

Далее описан процесс создания нового регуляризатора и добавления его в библиотеку. Данные действия подчиняются некоторым общим правилам, поэтому будут изложены в алгоритмическом виде:

1. В первую очередь требуется описать в файле `messages.proto` protobuf-сообщение — конфигурацию нового регуляризатора, после чего добавить соответствующий элемент в поле `Type` в сообщении-оболочке `RegularizerConfig`.

2. Затем необходимо написать `.h` и `.cc` файлы создаваемого регуляризатора и добавить их в проект `libartm`. В данном пункте следует учитывать несколько деталей, которые будут рассмотрены подробнее в конце данного раздела.

3. После нужно добавить в файл `c_interface.cc #include` с именем `.h` файла нового регуляризатора.

4. В `c_interface.cc` надо найти функцию `ArtmReconfigureRegularizer()`. В ней есть switch по типам регуляризаторов, в который требуется добавить case с типом своего регуляризатора (для этого можно просто скопировать один из существующих case и поменять в нём имена регуляризатора и его конфигурации).

5. Для того, чтобы использовать добавленный регуляризатор, осталось скомпилировать `messages.proto` (используя компилятор `protoc`) в файлы на C++ и Python, после чего перекомпилировать проекты `libartm` и `artm`.

Рекомендации по написанию файлов регуляризаторов. Любой регуляризатор, описанный в библиотеке, должен удовлетворять нескольким требованиям:

- Регуляризатор обязательно пишется (как и всё ядро библиотеки) на C++ (.cc и .h файлы).

- Регуляризатор представляет собой класс-наследник класса `RegularizerInterface`. Данный класс содержит два виртуальных метода, которые нужно описывать

[9] Данный раздел не имеет отношения к пользовательской документации и предназначен для разработчиков BigARTM. Описанная в нём информация требует более глубокого понимания устройства библиотеки

```
virtual bool RegularizeTheta(const Item& item,
                             std::vector<float>* n_dt,
                             int topic_size,
                             int inner_iter,
                             double tau)

virtual bool RegularizePhi(core::TopicModel* topic_model,
                           double tau)
```

Как понятно из названия, каждый метод отвечает за регуляризацию соответствующей матрицы.

- Класс регуляризатора вместе со всем своим содержимым должен быть описан в namespace `regularizer`.

- Технически допустима реализация в одном классе регуляризатора сразу для обеих матриц, однако рекомендуется этого избегать и описывать один (в математическом смысле) регуляризатор в двух разных классах и работать с ними как с разными объектами. Это существенно упростит конфигурацию отдельного регуляризатора, использование его данных в функциях регуляризации, а также избавит от необходимости задавать ненужные данные, когда потребуется регуляризация только одной матрицы.

- В случае, когда регуляризатору для работы требуются один или несколько словарей, необходимо учесть их количество и структуру, после чего зафиксировать и документировать эту информацию. Корректность работы регуляризатора будет гарантироваться только в случае, если подаваемые словари соответствуют предъявляемым требованиям.

- Файл `.h` будет отличаться для разных регуляризаторов только названием, достаточно скопировать какой-нибудь существующий (регуляризирующий ту же матрицу) и поменять необходимые имена типов.

- Файл `.cc` будет включать в себя реализацию соответствующего метода регуляризации. Сама реализация, в свою очередь, состоит из трёх концептуальных этапов — считывания данных из объекта-конфигурации, проверки корректности этих данных и собственно регуляризации. Считывание данных рекомендуется производить в структуры тех же типов, что и используемые в конфигурации — т.е. описанные в файлах `messages.pb.h` и `messages.pb.cc` protobuf-типы. Во время проверки корректности ввода следует возвращать из функции `false` в случае выявления несоответствия. Наконец, после окончания этапа регуляризации следует вернуть `true`. Кроме того, описываемый регуляризатор обязательно должен удовлетворять замечанию 6.3.2.

Замечание: В целом, при написании нового регуляризатора рекомендуется опираться на существующие реализации, это может сэкономить немало времени.

7 Эксперименты

В данном разделе будет рассмотрен модельный эксперимент с использованием регуляризаторов в BigARTM. Его задачей является демонстрация использования реализованного механизма регуляризаторов, от него не ожидается получения ответов на какие-либо исследовательские вопросы.

7.1 Описание эксперимента

Производится обучение двух тематических моделей — с регуляризатором и без. Для регуляризации модели используется регуляризатор сглаживания/разреживания (в данном эксперименте будем использовать только разреживание). Параметры эксперимента следующие:

1. Обучающая коллекция документов — NIPS ($\approx$ 1600 документов).
2. Объём словаря — $\approx$ 13000 терминов.
3. Число внешних итераций — 12, внутренних — 5.
4. Число процессоров — 2.
5. Число тем — 16

Регуляризация имеет следующую траекторию:

- 0 — 2 итерации: нет регуляризации;
- 3 — 4 итерации: на каждой внешней итерации от счётчиков n_{wt} отнимается 15;
- 5 — 6 итерации: на каждой внешней итерации от счётчиков n_{wt} отнимается 25;
- 7 — 8 итерации: на каждой внешней итерации от счётчиков n_{wt} отнимается 40;
- 9 — 11 итерации: на каждой внешней итерации от счётчиков n_{wt} отнимается 60;

7.2 Результаты эксперимента

Значимыми параметрами качества тематической модели являются различность тем, разреженность матриц Φ и Θ (без учёта фоновых тем), объём ядер тем (т.е. слов, характеризующих данную тему). Регуляризация имеет перед собой задачу оптимизировать именно эти величины, не ухудшив при этом переплексию по сравнению с нерегуляризованной моделью. В BigARTM на данный момент отсутствует механизм оценивания параметров матрицы Θ, поэтому ограничимся одним критерием — степенью разреженности матрицы Φ. Как видно из графиков на 1, регуляризация модели не изменила существенно перплексию на обучающей выборке. В то же время, разреженность матрицы Φ существенно повысилась, что очевидно из графиков на 2.

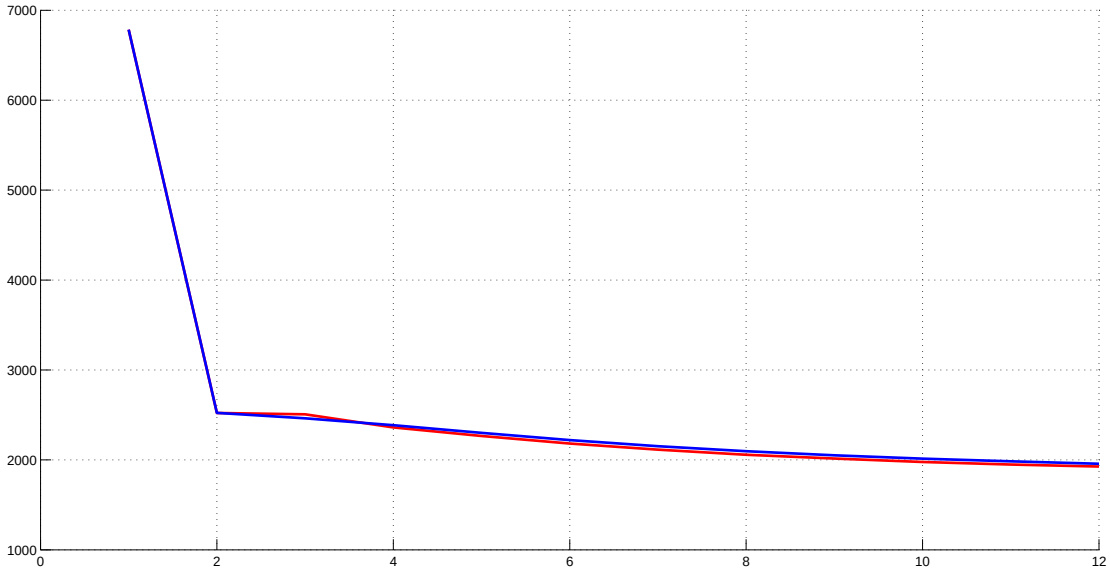


Рис. 1: Ось X — номер внешней итерации, ось Y — перплексия модели на обучающей выборке. Красная линия соответствует регуляризованной модели, синяя — нерегуляризованной

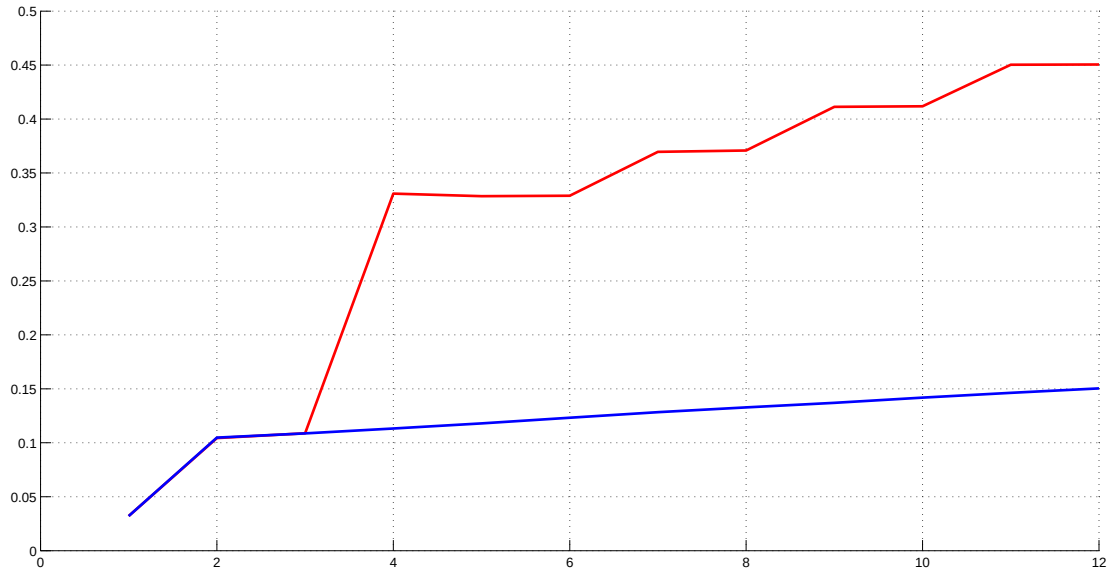


Рис. 2: Ось X — номер внешней итерации, ось Y — разреженность матрицы Φ в процентах от общего числа элементов. Красная линия соответствует регуляризованной модели, синяя — нерегуляризованной

8 Заключение

Все задачи, поставленные перед данной курсовой работой, были выполнены. Был произведён обзор текущих возможностей BigARTM, перспективы её развития, описаны инструкции по работе с библиотекой. Кратко были рассмотрены существующие библиотеки тематического моделирования, их алгоритмические и архитектурные особенности, в т.ч. и нашедшие применение в BigARTM.

В библиотеке BigARTM была создана и отлажена структура работы с регуляризаторами тематических моделей. Она была протестирована и доказала свою работоспособность. Были описаны примеры регуляризаторов, которые могут быть использованы как при решении реальных задач, так и в качестве основы для создания более сложных механизмов регуляризации.

Дальнейшая деятельность в рамках проекта по созданию библиотеки будет направлена на внедрение описанных в тексте работы планируемых нововведений.

Список литературы

[1] Воронцов К. В. Аддитивная регуляризация тематических моделей коллекций текстовых документов // Доклады РАН. 2014. — Т. 455., №3. 268–271.

[2] T. Hofmann Probabilistic latent semantic indexing // Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval. — New York, NY, USA: ACM, 1999. — Pp. 50–57.

[3] https://developers.google.com/protocol-buffers/docs/tutorials

[4] David Newman, Arthur Asuncion, Padhraic Smyth and Max Welling — Distributed Algorithms for Topic Models, 2009.

[5] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen and Edward Y.Chang — PLDA: Parallel Latent Dirichlet Allocation for Large-Scale Applications, 2009.

[6] A. J. Smola and S. Narayanamurthy — An architecture for parallel topic models, 2010.

[7] Liu, Z., Zhang, Y., Chang, E. Y., and Sun, M. — PLDA+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing, 2011.

[8] Воронцов К.В., Потапенко А.А. — Аддитивная регуляризация тематических моделей, 2014.